The International Conference on Dependable Systems and Networks (ICDSN/FTCS), 01-04 July 2001: Göteborg, Sweden.

Robustness Testing and Hardening of CORBA ORB Implementations

Jiantao Pan, Philip Koopman & Daniel Siewiorek
ECE Department & ICES
Carnegie Mellon University
Pittsburgh, PA, USA
{jpan,koopman}@cmu.edu
dps@cs.cmu.edu

Yennun Huang*
PreCache Inc.
555 Madison Ave.
New York, NY, USA
yennunhuang@yahoo.com

Robert Gruber
AT&T Labs – Research
180 Park Ave.
Florham Park, NJ, USA
gruber@research.att.com

Mimi Ling Jiang
4733 Maripoe St.
Pittsburgh,
PA, USA
mjiang@cmu.edu

Abstract

Before using CORBA (Common Object Request Broker Architecture) applications in mission-critical scenarios, it is important to understand the robustness of the Object Request Broker (ORB) being used, which forms the platform for CORBA applications. We have extended the Ballista software testing technique to test the exception-handling robustness of C++ ORB client-side application interfaces, and have tested two major versions of three ORB implementations on two operating systems, yielding robustness failure rates ranging from 26% to 42%. To improve ORB robustness, we also propose a probing method to harden object and pseudo-object related data types against exceptional inputs. Using these probes on omniORB 2.8 has proven to be effective in eliminating some cases of robustness failures found during testing. These results suggest that CORBA implementations currently have significant robustness vulnerabilities, but that some important classes of problems can be overcome with better exception-handling approaches.

1. Introduction

The development of CORBA (Common Object Request Broker Architecture) has advanced the concept of component software: diverse software modules implemented in different programming languages can be integrated as a distributed system using the CORBA interface, and interact in a plug-and-play manner. Using this new component software model, an application can be built by assembling legacy software modules, third-party software modules, and custom-made software modules on a common CORBA platform connected by ORBs (Object Request Brokers), instead of developing a totally custom-made monolithic application, saving both development cost and time to market.

Many mission-critical systems, such as aerospace/defense, banking/finance, healthcare/insurance, e-commerce and telecommunication applications [15], have selected a distributed architecture based upon CORBA. Enterprises and government agencies from all over the world, including NASA, Boeing, Chase Manhattan bank, Motorola, Ericsson, and Independence Blue-Cross, are using CORBA in various applications, from Web-based online banking to cellular phone management, patient care and even applications for the Hubble Space Telescope. Recently, the Object Management Group (OMG) initiated a Space Domain Task Force to encourage the Space and Satellite industry to foster the emergence of cost effective, timely, commercially available and interoperable space, satellite, and ground system domain software components through CORBA technology [17]. While cost and development time is a common consideration for general purpose systems, the robustness of the software – the degree to which a software component functions correctly in the presence of invalid inputs or stressful environmental conditions [5] – is almost always a major concern for mission-critical applications as the examples listed above. It is important that these applications are resistant to failures caused by abnormal inputs.

CORBA applications used in critical scenarios must be robust. But, the heterogeneous environment; the use and reuse of commercial off-the-shelf, third-party and legacy software modules; and their complex interactions will all be likely to trigger exceptions. Thus, the graceful handling of expected and unexpected exceptions is critical for the robustness of CORBA-based systems.

CORBA applications are built upon an Object Request Broker (ORB) interface. The ORB accepts requests from CORBA applications, processes the requests, and manages the communication among different objects, applications and ORBs. The robustness of the ORB is very important since the ORB is the operating platform of CORBA applications and the venue for a CORBA software component to communicate and interact with the rest of the system and the outside world. Developers of critical applications will often need to know the robustness of candidate ORB implementations prior to deciding which one to use. However, methods to evaluate CORBA ORB robustness are rare.

* This work was completed when the author was in AT&T Labs.

There is an urgent need for a method to evaluate the robustness of ORB implementations.

This paper makes three contributions. First, this paper quantitatively measures and compares the exception-handling robustness of CORBA ORB implementations using the Ballista robustness testing methodology [9]. Second, some common exception-handling robustness problems of the ORB implementations under test are identified. Third, methods to improve the robustness of ORB exception-handling robustness are proposed and evaluated. A version of the Ballista testing tool has been implemented for testing C++ ORBs and used to study two major versions of three ORB products on two operating systems. The operations tested are selected from CORBA 2.1 standard [18]. In the text that follows, Section 2 details the testing methodology, Section 3 discusses the experimental setup, Section 4 gives results, analysis and suggestions on improving ORB robustness, and Section 5 lists related work. Conclusions can be found in Section 6.

Table 1. Classification of robust and non-robust behaviors.

Robust behaviors	**Successful return (no exceptions)** **Raise CORBA exception**
Non-robust behaviors (Robustness failures)	**Computer crash (Catastrophic failure)** **Thread hang (Restart failure)** **Thread abort (Abort failure)** **Raise unknown exception** False success (Silent failure) Misleading error information (Hindering failure)

2. Methodology

Many factors can contribute to the robustness of a software component. Although stressful environmental conditions are important, we focus on measuring how gracefully a software module under test behaves under exceptional inputs. Typically, two methods are used to handle exceptional input situations: returning error-return codes and raising exceptions. Error-return codes are used extensively in software implemented using the C language, such as POSIX standard [6] functions and system calls in many operating systems. Raising exceptions is used as the standard reporting and handling mechanism for exceptional inputs in the CORBA standard [18] for C++ and Java mappings. In this study, we test and measure the exception-handling robustness of C++ ORB implementations.

2.1 Metric

Previous Ballista work [9] proposed a way to measure the robustness of software modules. In this work we extend the same approach to measure CORBA ORB robustness by testing API calls.

The CORBA standard defines a common API for ORB vendors to implement. This API defines a collection of operations that a client or server object can request the ORB to perform on behalf of a user program. The CORBA standard has no restrictions on how the vendors should implement the operations specified in the standard. However, it does have requirements on how an ORB should perform under abnormal input situations:

> "The ORB manages the control transfer and data transfer to the object implementation and back to the client. In the event that the ORB cannot complete the invocation, an exception response is provided." [18]
>
> "If an abnormal condition occurs during the performance of a request, an exception is returned." [18]

The above excerpts from the CORBA standard clearly state that an ORB operation is robust under exceptional inputs if the operation can identify the exceptional inputs and raise exceptions. However, the exceptions should be defined and reflect the actual exceptional situations. We consider raising unknown exceptions to be non-robust, because no useful information is given for error recovery and there is no guarantee that the ORB is in a consistent state when an unknown exception is thrown.

Table 1 lists the possible robust behaviors and non-robust behaviors that may occur in testing ORB implementations, and maps them to the existing Ballista CRASH scale metric [9]. Among the listed robustness failures, computer crashes, thread hangs, thread aborts and unknown exceptions can be automatically detected by Ballista. False successes and misleading error returns cannot be discovered in an automated manner using the Ballista harness and are not measured in this study. More discussion about false successes (*i.e.* silent failures) and how to estimate them can be found in [9]. In summary, the items highlighted in boldface in Table 1 are the responses that we expect to find in testing the robustness of CORBA ORB implementations.

2.2 CORBA ORB robustness testing architecture

We adapted the Ballista software robustness testing methodology to evaluate ORB implementation robustness. The Ballista testing framework is designed to test COTS (Commercial Off-The-Shelf) software modules for exception-handling robustness problems triggered by invalid inputs.

Figure 1 shows the CORBA ORB testing architecture using Ballista. The *Ballista server* performs client code generation and test case generation. The *test manager* of the Ballista client iterates through test cases and manages test

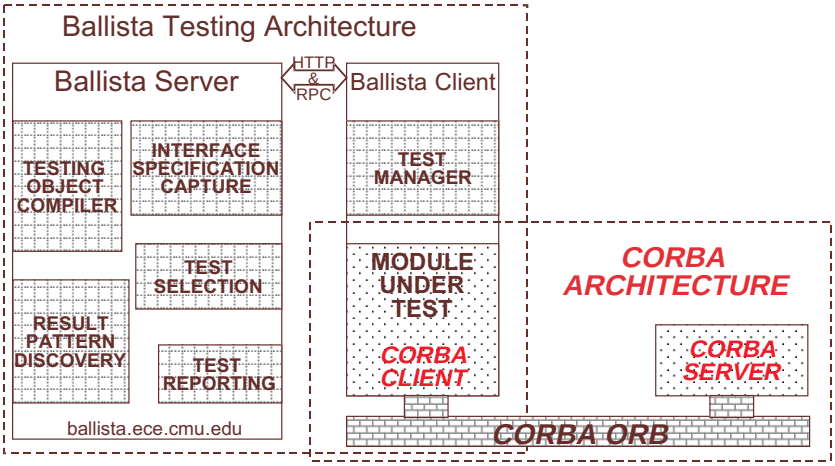


Figure 1. CORBA testing architecture using Ballista.

case set-up, response monitoring and test case cleanup. The *module-under-test* in the Ballista architecture is in this case a CORBA client. In testing, the *module-under-test* communicates and interacts as necessary with the *CORBA server* object via the *CORBA ORB* interface.

For each test case, the *test manager* spawns a corresponding *module-under-test* thread, and monitors the status of this child thread. Figure 2 shows a generic *module-under-test* in pseudo-code form. The *initialization()* part initiates the ORB and creates necessary variables to be used during the testing process. The *parameter_instantiation(parameter_list)* procedure creates an instance of each parameter from the values specified in the test case database. The actual call to the operation under test appears in *ORB_operation_invocation(parameter_list)*.

```
module-under-test{

  initialization();
  parameter_instantiation(parameter_list);
  ORB_operation_invocation(parameter_list);
  ...
  exception_handling();

}
```

Figure 2. module-under-test **pseudo-code.**

```
//userCatches
catch (const CORBA::SystemException& se)
catch (const CORBA::Exception& e)

//low-level exception catches
catch (...)
```

Figure 3. Exception catching levels.

Exceptions thrown by the ORB operation during testing are caught and analyzed by the *exception_handling()* section.

The different levels of exception handling are shown in Figure 3. Based on the CORBA standard, CORBA-defined exceptions are caught and categorized into `CORBA::SystemException` and `CORBA::Exception`. If the ORB operation under test only raises these CORBA-defined exceptions, it is considered robust. The *low-level exception catching* makes sure all other unknown exceptions which are not defined in CORBA standard are caught, which are classified as robustness failures.

2.3 Test case inheritance

A test case inheritance scheme is used to maximize the reuse of test cases. Most ORB operations use CORBA specific data types as parameters. For C++ mappings, the CORBA specific data types are eventually mapped to C++ language data structures. `CORBA::Flags`, for example, is mapped to `unsigned long` in the C++ language. We have designed an inheritance hierarchy to structure CORBA data types. A child data type inherits test cases defined in its parent data type and expands the parent data type by providing test cases specific to the child. As a general rule, a child data type usually expands its parent data type in value range or semantics. In the example inheritance tree shown in Figure 4, data type `CORBA::Flags` inherits all test cases (e.g. `MAX_UNSIGNED_LONG`) defined in the parent data type `unsigned long` (which also inherits test cases from its parent data type `unsigned int`), and adds `ARG_IN`, `ARG_OUT`, `ARG_INOUT`, *etc.*, as its specific test cases. These test cases for CORBA related data types have been selected based on the CORBA specification.

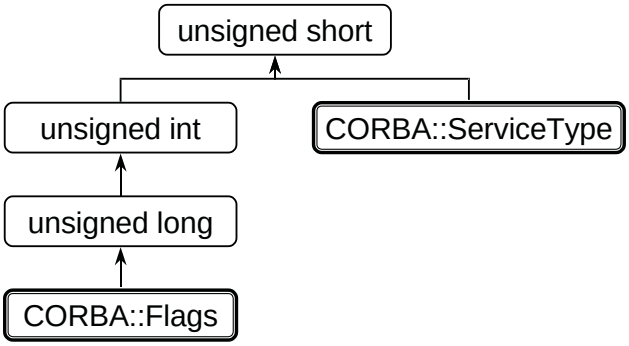


Figure 4. Sample test case inheritance.

3. Experimental setup

The Ballista CORBA client has been implemented for two major versions of three ORB implementations for the C++ language mapping on Solaris and Linux platforms.

3.1 ORB platforms under test

There are many ORB implementations available, forming a potential rich set of candidates to conducting our study. We chose Orbix, omniORB and VisiBroker as the candidate platforms based on popularity and availability. Specifically, the following ORBs were tested:

- Orbix 3.0.1 and Orbix 2000
- omniORB 2.8 and omniORB 3.0
- VisiBroker 3.3 and VisiBroker 4.0

Orbix and VisiBroker are commercial implementations, while omniORB is freely available under GNU public license. Orbix 2000 and VisiBroker 4.0 were tested using evaluation downloads from the vendor web sites, while earlier versions were tested using existing licensed copies. All ORBs were tested on a Sparc workstation running Solaris 5.6 to facilitate fair comparisons. Orbix 2000, omniORB 3.0 and VisiBroker 4.0 were also tested on a Pentium machine running RedHat Linux 6.2 (kernel version 2.2.14-5.0smp). The earlier versions of ORBs are not tested on the Linux platform because VisiBroker 3.3 and Orbix 3.0.1 do not have publicly available Linux releases.

3.2 Test set

A subset of basic ORB operations defined in CORBA standard 2.1 [18] was chosen as the test set. The test set includes operations defined in basic interfaces, such as most operations in interfaces `CORBA::Request`, `CORBA::NVList`, `CORBA::Context`, `CORBA::ORB` and `CORBA::Object`, which appear in Chapter 4, 5, and 6 of the CORBA standard 2.1. These interfaces are generally considered important because most CORBA programs will have to use them directly, or have to inherit from those interfaces. For example, `CORBA::ORB_init` has to be called by both the client side and server side to initialize the ORB for any CORBA 2.1 programs. All CORBA objects defined in CORBA 2.1 programs must inherit the base object `CORBA::Object` directly or indirectly. `CORBA::Context` captures important information about the client, environment, or circumstances of a request.

Most of the operations under test are client-ORB interactions, not client-server operations or inter-ORB operations. No GIOP/IIOP operations have been tested. However, the test set could be expanded in the future to include inter-ORB client-server calls and other new operations defined in later standards, such as POA operations in CORBA standard 2.3 supported by Orbix 2000, omniORB 3.0 and VisiBroker 4.0.

3.3 Implementation issues

Although Orbix 3.0.1, omniORB 2.8 and VisiBroker 3.3 all claim to support or fully comply with CORBA standard 2.1, and the advanced versions claim to be compatible with CORBA standard 2.3, not all operations in the CORBA standard are supported by every ORB. This is partly because of rapid updates to and ambiguities within the CORBA 2.1 standard. For example, we have observed that the same operations may appear under different names on different ORBs: `CORBA::Object` operation `get_policy()` is defined as `CORBA::Object::_get_policy()` in Orbix [8], appears as `CORBA::Object::get_policy()` in VisiBroker [7], and is undefined in omniORB [11].

Some operations, such as `get_default_context()` and `get_service_information()` for Orbix 2000, have prototypes defined but are not implemented. They always raise CORBA system exception `CORBA::NO_IMPLEMENT` during testing. Although this response is valid per the CORBA standard, it is unfair to compare these operations (which technically would be 100% robust) with implemented versions from other ORBs (which will likely have failures). Therefore, the operations without implementations are deleted from Orbix 2000 test sets.

Due to the above issues, the test operations actually launched for each ORB are not fully identical. But a reasonable comparison can still be made by taking averages of all the operations tested for each ORB, mitigating this effect.

4. Experimental results and analysis

4.1 Overview

Table 2 summarizes the results for the ORBs under test. The total number of test cases and the total number of operations tested are given. For example, for the omniORB 2.8 Solaris build, there are 6999 test cases launched for 22 operations, within which one operation `create_list()` exhibits thread hangs, 17 operations trigger thread aborts, and one operation `CORBA::string_alloc()` raises unknown exceptions.

We have observed another failure mode while testing the ORBs, other than the common robustness failures listed in Table 1. While testing operation `create_list()` for omniORB 2.8 and omniORB 3.0 on Sun Solaris platform, we have found a *libthread panic* failure. The failure is denoted by a * in Table 2. This failure cannot be isolated to

Table 2. Overview of ORB robustness testing results.

Product	Version	Total test cases	Total functions tested	Number(#) of functions exhibiting each type of failure					
				Solaris 5.6			RedHat Linux kernel version 2.2.14-5.0smp		
				# of functions with thread hangs	# of functions with thread aborts	# of functions with unknown exceptions	# of functions with thread hangs	# of functions with thread aborts	# of functions with unknown exceptions
Orbix	3.0.1	5361	22	6	16	0	N/A	N/A	N/A
omniORB	2.8*	6999	22	1	17	1	N/A	N/A	N/A
VisiBroker	3.3	6581	21	1	17	0	N/A	N/A	N/A
Orbix	2000	3219	17	0	14	1	0	14	1
omniORB	3.0*	6581	21	2	17	1	1	17	1
VisiBroker	4.0	7023	23	7	20	0	1	21	0

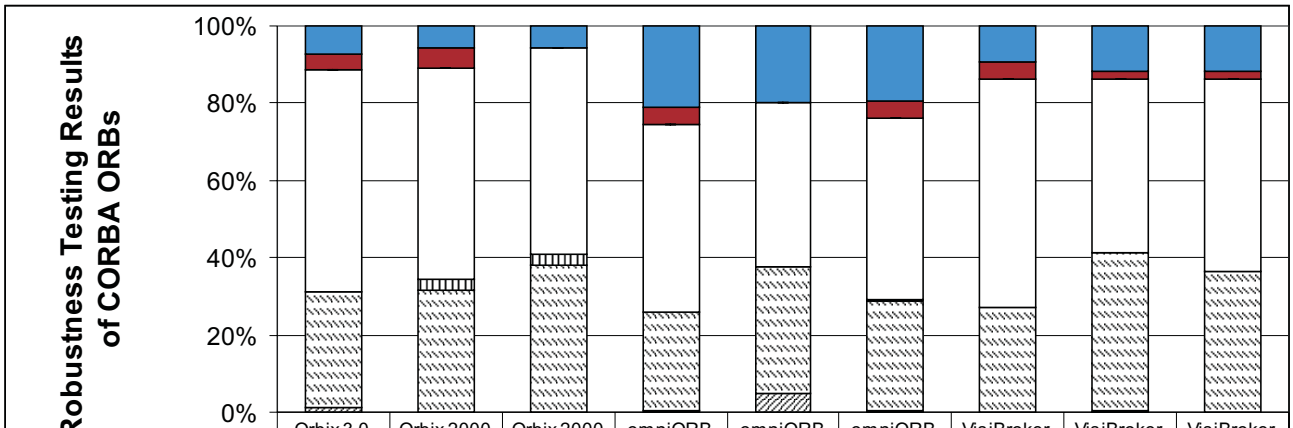
* libthread panic observed

one test case because it is related to resource problems; however, it does happen each time `create_list()` is tested. One possible explanation is that when the testing thread times out and is killed when a thread-hang failure happens, *libthread* has a resource leakage that eventually leads to a panic. This failure may be unrelated to the ORB under test and is not counted in the final results.

4.2 Average percentage of failures

Results are analyzed using a straight average across the failure rate for all the operations as a comparison metric. Figure 5 shows the robustness testing results for the ORBs we have tested. Each bar represents an ORB implementation, with the average percentage of robustness failures (including thread aborts, thread hangs and unknown exceptions) shown at the bottom, and the average percentage of robust behaviors (including `CORBA::SystemException`, `CORBA::Exception` and no-exception responses) shown at the top. Figure 5 shows that all ORB implementations studied have a high thread-abort percentage under the current test set, ranging from 25.44% for the omniORB 2.8 Solaris build to 41.02% for the VisiBroker 4.0 Solaris build. Thread-hang failures are less common and usually concentrate in only a few operations such as `create_list()`. Orbix 2000 has more unknown exceptions than other ORB implementations but is free of thread hangs. No unknown exceptions are found in the VisiBroker group.

The omniORB group demonstrates a higher percentage of CORBA exceptions, indicating a better exception-han-



	Orbix 3.0 Sun	Orbix 2000 Sun	Orbix 2000 Linux	omniORB 2.8 Sun	omniORB 3.0 Sun	omniORB 3.0 Linux	VisiBroker 3.3 Sun	VisiBroker 4.0 Sun	VisiBroker 4.0 Linux
CORBA::SystemException%	7.31%	5.67%	5.51%	20.94%	19.81%	19.59%	9.50%	11.63%	11.63%
CORBA::Exception%	4.10%	5.05%	0.33%	4.58%	0.13%	4.23%	4.36%	2.14%	2.27%
No exception%	57.28%	54.73%	53.08%	48.54%	42.26%	47.16%	59.13%	44.73%	49.72%
Unknown exception%	0.00%	2.88%	3.00%	0.21%	0.20%	0.20%	0.00%	0.00%	0.00%
Thread abort%	30.11%	31.66%	38.08%	25.44%	32.93%	28.51%	26.88%	41.02%	36.32%
Thread hang%	1.21%	0.00%	0.00%	0.29%	4.66%	0.30%	0.13%	0.47%	0.06%

Figure 5. Robustness testing results for 2 versions of 3 ORB products on 2 operating systems for a collection of CORBA 2.1 standard operations. Each bar represents the average percentage of different response categories of approximately 22 functions tested using 3000 to 7000 test cases.

dling scheme. OmniORB 2.8 has the highest robust exception percentage, at over 25%.

4.3 Discussion

As shown in Figure 5, the results show a noticeable, sometimes significant, increase in average percentage of robustness failures from an older version of a product to a new version. Similar phenomena have also been observed in previous Ballista testing results on the HLA RTI simulation backplane [3]. In POSIX testing [10], two operating systems had an increase in robustness failure rate going from older versions to newer versions, while three operating systems did the opposite. One possible explanation is that adding significant amounts of new code to implement new features in a new software release can create robustness failures until problems are fixed based on field defect reports. If a version change is more evolutionary, then it is more likely that robustness will improve with a new release. In the case of CORBA ORBs, fast evolution and frequent updates to the CORBA standard would seem likely to force addition of significant amounts of new code into each release, leading to a likely increase in robustness failure rates as shown in Figure 5.

The differences in results between operating systems are not marked enough to make definitive statements about how much operating systems factor into ORB robustness. For Orbix 2000, the Solaris build is more robust than the Linux version, while for omniORB 3.0 and VisiBroker 4.0, the results are the opposite. However, these differences do indicate that it is the OS/ORB pair that determine overall robustness rather than robustness being solely a property of the ORB selected.

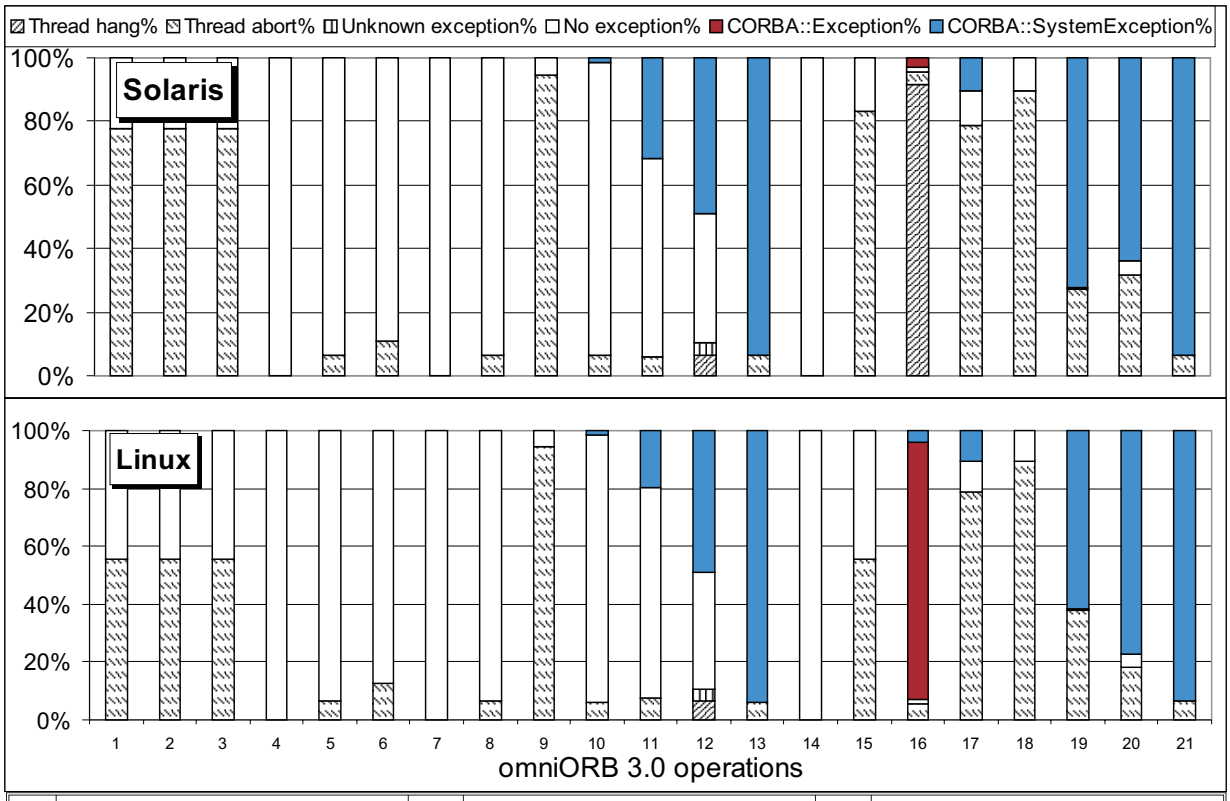
Figure 6 gives a snapshot of the testing results for omniORB 3.0 on Solaris and Linux platforms, where we can see that most operations have the same failure profile across platforms, with only a few exceptions. Other than the *libthread panic* failure observed on Solaris, the most different testing result was found for operation `resolve_initial_references()`, which had a significant percentage of thread hangs on Solaris but only a small percentage of thread-abort failures on Linux. The cause of this difference is currently unknown.

The operations that have a failure rate higher than 50% are as follows: `_duplicate()`, `CORBA::is_nil()`, `CORBA::release()`, `_is_equivalent()`, `object_to_string()`, `send_multiple_requests_deferred()`, and `send_multiple_requests_oneway()`. Among these seven operations, the first five take `CORBA::Object` as one of the parameters. The last two take `RequestSeq`, which is a sequence of object references. This indicates that CORBA object and pseudo-object data type implementations have more prevalent robustness vulnerabilities than other data types tested for omniORB implementations.

An interesting observation is that for both operating system platforms, operations `_duplicate()`, `CORBA::is_nil()`, and `CORBA::release()` have exactly the same failure profile, and they all take a single parameter typed `CORBA::Object`. It appears that the failure profile is correlated with parameter types if the parameter types are CORBA object references, or object references in general. It is possible that these failures are caused by methods associated with objects, especially if one presumes that different CORBA operations employ the same access patterns for object reference data types: calling member functions to access private data structures, using constructors to create object instances, using destructors to deallocate objects, *etc*.

The testing results in previous sections have not shown us a promising picture of ORB implementation robustness, although the CORBA specification explicitly requires robust responses. On the other hand, for the ORBs we have tested, we have observed that some important and obvious exceptional values, such as NULL value for pointers and references, are handled by most ORB vendors. In most cases, a `CORBA::BAD_PARAM` exception is seen when a NULL parameter is used improperly. Compared to some POSIX operating systems where NULL causes a significant rate of robustness failures for many functions [10], this is certainly an advancement. However, for CORBA applications to achieve the level of robustness desirable for critical applications, being able to handle simple and obvious cases, such as NULL as pointer and reference values, or zero as divisors, is not sufficient. There are large classes of more subtle but realistic exceptional input values, *e.g.,* out-of-range values, uninitialized data structures, erroneously released objects, and corrupted data structures. Most ORB operations we have tested failed to identify these exceptional test values gracefully by throwing a `CORBA::BAD_PARAM` exception.

Exception handling is often skimped upon for two main reasons. First, comprehensive exception handling is perceived as costing performance. Second, it is labor-intensive to develop individual parameter checks against exceptional inputs on a case-by-case basis. As a simple solution to the first problem, ORB vendors can provide the user with an infrastructure that has two different operating modes, or with standard exceptional-input checking operations for critical scenarios. For the second problem, instead of developing individual parameter checks, we can use more advanced techniques such as program assertions [19] and data structure signatures [21] to check run-time data consistency and integrity under a CORBA and object-oriented context. The following section demonstrates that it is possible to provide the user a means to protect against a



1	CORBA::Object::_duplicate	8	_is_a	15	object_to_string
2	CORBA::is_nil	9	_is_equivalent	16	resolve_initial_references
3	CORBA::release	10	add_item	17	send_multiple_requests_deferred
4	CORBA::string_alloc	11	create_child	18	send_multiple_requests_oneway
5	CORBA::string_dup	12	create_list	19	set_one_value
6	CORBA::string_free	13	delete_values	20	set_values
7	_hash	14	get_default_context	21	string_to_object

Figure 6. OmniORB 3.0 robustness testing results for selected CORBA functions on Solaris and Linux platforms. Each bar represents one CORBA operation whose name is listed in the table.

large class of exceptional input values for object data types at a reasonable cost, using a simple method combining assertions and data structure signatures.

4.4 Robustness failure protection for object reference data types

We propose a probe technique that is flexible and can be easily implemented by ORB vendors to protect against a large class of robustness failures, especially those failures caused by invalid and uninitialized object and pseudo-object references. A probe function is a function that can be used to determine the validity of a parameter value. There are three important characteristics for a probe function: sensitivity, non-intrusiveness and robustness. The probe function should be designed to be as sensitive as possible so that it can discern an invalid object or pseudo-object reference value from a valid one. It must be non-intrusive so that the parameter value as well as the state of the program remains unchanged after the check. It must be robust so that no extra robustness failures are introduced by the probe function itself. The probe function should also be as lightweight as possible to minimize performance overhead; however, there may be a trade-off between sensitivity and performance.

We have found that omniORB 2.8 and omniORB 3.0 actually provides the necessary basis for a simple probe technique. In omniORB 2.8 and 3.0, each object and pseudo-object data type is assigned an undocumented sequence number named `PR_magic`. This "magic" number

is unique for each object and pseudo-object data type and serves as an identity mark. When an instance of an object or pseudo-object data type has been correctly set up, the constructor initializes a member variable `pd_magic` to contain the correct `PR_magic` value specific to this data type, its "identity mark". This variable is set to "invalid" by the destructor when the reference is freed. Therefore, a valid reference which is properly initialized will contain a correct magic number set up in the variable `pd_magic` during its lifetime. If `pd_magic` does not contain the correct value, the reference must be invalid. Therefore, we can detect invalid and uninitialized references to objects and pseudo-objects by checking whether `pd_magic` contains the right `PR_magic` value at run time. An undocumented static member function `PR_is_valid()` is defined in omniORB for each object and pseudo-object data type to do this checking. It is possible that a valid `PR_magic` value could be stored in an uninitialized object by chance, but for most situations involving uninitialized memory values this is expected to be unlikely.

Unfortunately, the above functionality provided by omniORB is not robust enough to serve as a probe function for our purposes. For many invalid object references that are in our test set, the `PR_is_valid()` check triggers a robustness failure instead of returning false by attempting to read the `PR_magic` value from an illegal memory address. Also, NULL object references cannot be detected by `PR_is_valid()` checking.

We have taken several steps to create a sensitive, robust and non-intrusive probe functionality by refining the `PR_is_valid()` checking method. The sensitivity of `PR_is_valid()` is made largely acceptable by adding a call to `CORBA::is_nil()`, the specialized NULL object reference checking method defined in the CORBA standard. We have made the checking procedure more robust by adding necessary signal-handling code. A signal triggered while either `PR_is_valid()` or `CORBA::is_nil()` is accessing the parameter value also indicates that an invalid parameter value is detected. A multi-threaded checking scheme could also be used instead of the signal-handling method, but would probably have higher performance cost without significant optimization efforts.

We have conducted some initial experiments to study the effectiveness of this method. From our CORBA 2.1 operation test set, we selected a subgroup of eight operations that take CORBA object references or CORBA pseudo-object references as parameters. A simple protection-code generator was implemented to generate protection code automatically that is suitable for some parameter types. First, a NULL-checking experiment was conducted. Second, the `PR_is_valid()` checking code was generated and added to the target module. Third, a signal handler was installed. The result was were tested on omniORB 2.8 using the Linux platform.

Table 3. Effectiveness of probe hardening for 8 sample operations in the omniORB 2.8 Linux build.

	Thread-abort percentage
Original failure percentage	37.77%
After CORBA::is_nil() checking (alone)	42.45%
After CORBA::is_nil()+PR_is_valid checking	41.99%
After adding signal handling	0%

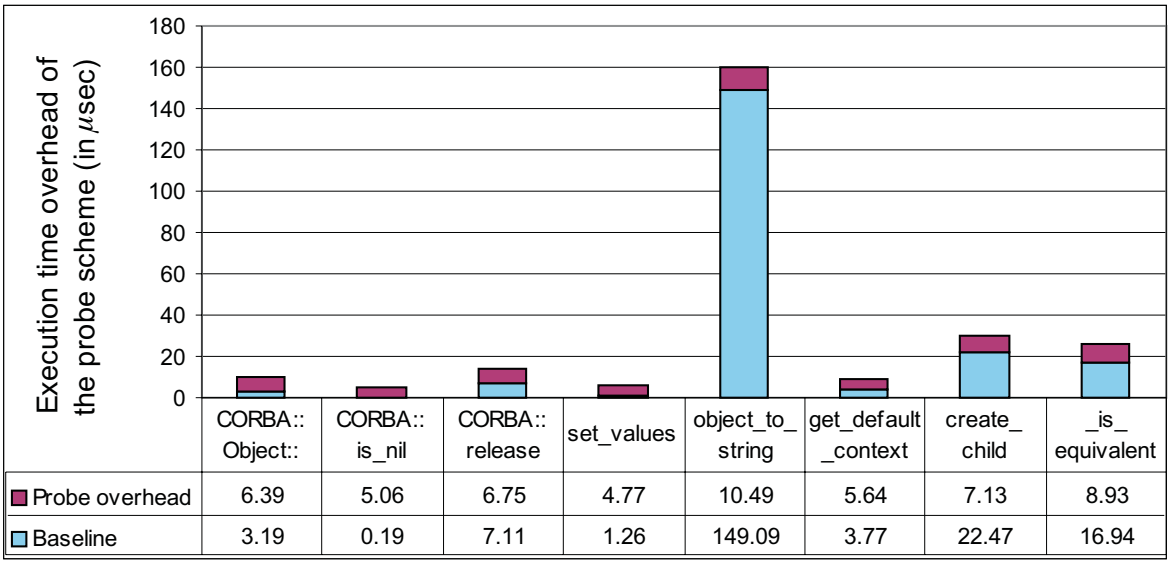
The results in Table 3 show that the protection scheme is effective. For the eight functions tested, all thread aborts, formerly 37.77% of the test cases, were successfully eliminated. Note that without necessary signal-handling mechanisms, `CORBA::is_nil()` and `PR_is_valid()` checking actually introduced additional robustness failures because none of the checking methods had a zero robustness failure rate.

The above results, however, do not mean that the probing method is always effective to detect all possible kinds of exceptional input values. Sometimes, an invalid object reference might by chance have a correct `pd_magic` value. This could happen when an object reference is correctly set up through the constructor, but later part of the object gets corrupted or overwritten while the `pd_magic` value remains intact.

Performance overhead was measured by running the target operation 5,000,000 times, with the probe functionality turned on or off, and calculating the difference of the average execution times. Valid parameter values are used in this measurement because it is more important to know the performance cost under normal execution situations. Since the protection code is compact and the test program is also very short, all instructions are expected to be resident in cache, giving an optimistic performance assessment.

The measurement results are shown in Figure 7. We can see that the probe function takes from 4.77 μs to 10.49 μs to execute, which is as high as 26 times the execution time for a simple operation `CORBA::is_nil()` (which contains only one `if` statement) and as low as 7% for a complex operation `object_to_string()`. This can be explained by the fact that the probe functions for all the object data types and pseudo-object data types use a uniform structure.

Although probe functions incur overhead, the overhead is likely to be negligible in real CORBA programs. This is because probes are static methods declared and run locally, and the bottleneck on CORBA platforms is usually the marshalling, unmarshalling and communication of data and methods over the network.



	CORBA:: Object::	CORBA:: is_nil	CORBA:: release	set_values	object_to_ string	get_default _context	create_ child	_is_ equivalent
Probe overhead	6.39	5.06	6.75	4.77	10.49	5.64	7.13	8.93
Baseline	3.19	0.19	7.11	1.26	149.09	3.77	22.47	16.94

Figure 7. Performance overhead of the probe scheme.

From this experiment we see that a simple probing technique can protect references to object data types and pseudo-object data types against a class of exceptional values. Similar methods might be standardized and generated as part of stub code by the `CORBA idl` compiler, so that users can use it to protect their custom data types selectively. Future work will include extending this method to protect against non-object data types with minimum or no compiler support.

5. Related work

Most previous work comparing ORB implementations has focused on measuring and optimizing ORB performance. [4] measures latency and throughput of Orbix and VisiBroker over high-speed ATM networks and identifies major overhead sources. The study in [14] provides a performance pattern language and a performance measurement object that can be used to extensively test ORB performance. The CORBA comparison project [2] compares omniORB, ORBacus, and Orbix using a rich set of benchmarks, mainly focusing on latency, throughput and scalability. Robustness of these ORBs is also briefly compared in terms of their maximum message size and the number of objects they can handle.

Fault injection on Orbix and DCOM applications [1] studies distributed object behavior under real and simulated failure scenarios. Failures at thread-level, process-level, and machine-level are simulated and injected into the server, and the response of the client is monitored and categorized, which marks the difference from our work. Our approach tries to manifest robustness failures in the CORBA ORB native code using exceptional inputs.

Various efforts have been made to build fault-tolerant CORBA applications, CORBA services and middleware, such as [12][13][20]. The Fault-Tolerant CORBA Standard [16] extends CORBA for applications requiring high dependability, attempting to eliminate sources of single-point failures. This standard mainly aims at tolerating crash failures using replication and does not address issues of exceptional parameter input handling.

Previous Ballista testing of the High Level Architecture Run-Time Infrastructure (HLA RTI) [3] provides another example of applying the Ballista testing methodology to testing distributed applications. The RTI is a standard distributed simulation system intended to provide completely robust exception handling. This effort extended the Ballista architecture for testing exception-based error reporting models and object-oriented software structures, which paves the path for the work presented here.

The general idea of the probe technique falls into the category of executable assertions introduced as early as [19]. Executable assertions are executable statements made about the variables in a program. If these statements do not hold true, an error has occurred.

The probe technique is also a variant of the data-structure signature technique [21], which uses a modified compiler to embed a signature in front of data structures to detect data access faults. Performance overhead can be largely minimized by using a special signature monitor that can be added to a standard pipeline processor.

6. Conclusions

In this paper we have introduced a methodology to test and measure the exception-handling robustness of CORBA ORB implementations using Ballista. We have ported Ballista testing clients to work with the ORBs, and tested two major versions of three ORB implementations on two operating systems for several CORBA 2.1 standard operations. This approach enables us to evaluate the robustness of specific ORB implementations, to compare different ORB implementations provided by various vendors, and to enhance the robustness of a specific ORB implementation.

We have presented results on the average percentage of failures for up to 23 functions per ORB. Testing for exceptional parameter value handling capabilities has revealed normalized average robustness failure rates ranging from 26% to 42%. These results suggest that users must pay close attention to ORB robustness issues when building critical applications on CORBA-based systems.

To improve ORB robustness, we propose a probe technique that checks for the presence of a special value that is stored in data structures upon initialization. A prototype implementation of this technique using capabilities already built into omniORB demonstrated that this approach can be effective at eliminating robustness failures for some data types.

Future work includes expanding robustness testing to include CORBA servers as well as clients, and creating a wider variety of techniques for robustness improvement of CORBA data types.

7. Acknowledgments

The authors would like to thank Chia-Yen Shih and Christof Fetzer from AT&T Labs for their support. This particular work was funded primarily by AT&T Labs – Research. The Ballista project is supported in part by DARPA (contract DABT63-96-C-0064).

8. References

[1] P. Emerald Chung, Woei-Jyh Lee, Joanne Shih, Shalini Yajnik and Yennun Huang, “Fault-Injection Experiments for Distributed Objects”, Proceedings of the International Symposium on Distributed Objects and Applications, Edinburgh, United Kingdom, September 5 - 7, 1999.

[2] Distributed Systems Research Group, “CORBA Comparison Project”, http://nenya.ms.mff.cuni.cz/thegroup/COMP. Accessed June 10, 2000.

[3] Kimberly Fernsler and Philip Koopman, “Robustness Testing of A Distributed Simulation Backplane”, Proceedings of ISSRE 99, Boca Raton, Florida, November 1-4, 1999.

[4] Aniruddha Gokhalé and Douglas Schmidt, “Evaluating CORBA Latency and Scalability Over High-Speed ATM Networks”, Proceedings of ICDCS 97, Baltimore, MD, May 27-30, 1997.

[5] IEEE Computer Society, “IEEE Standard Glossary of Software Engineering Terminology (IEEE Std 610.12-1990)”, Dec. 10, 1990.

[6] IEEE Computer Society, “IEEE Standard for Information Technology – Portable Operating System Interface (POSIX) — Part 1: System Application Program Interface (API) – Amendment 1: Real-time Extension [C Language] (IEEE Std 1003.1b-1993)”, 1994.

[7] Inprise Corporation, “VisiBroker for C++ Reference, Version 4.0”, 2000.

[8] IONA technologies PLC, “Orbix 2000 Programmer’s Reference”, March 2000.

[9] Philip Koopman & John DeVale, “The Exception Handling Effectiveness of POSIX Operating Systems”, IEEE Transactions on Software Engineering, September 2000.

[10] Philip Koopman and John DeVale, “Comparing the Robustness of POSIX Operating Systems”, Proceedings of FTCS 29, Madison, Wisconsin, June 15-18, 1999.

[11] Sai-Lai Lo and David Riddoch, “The omniORB2 version 2.8 User’s Guide”, AT&T Laboratories Cambridge, July 1, 1999.

[12] Silvano Maffeis, “A Fault-Tolerant CORBA Name Server”, Proceedings of the 1996 IEEE Symposium on Reliable Distributed Systems. Niagara-on-the-Lake, Canada: IEEE, October 1996.

[13] L. Moser, P. Melliar-Smith and P. Narasimhan, “A Fault Tolerance Framework for CORBA”. Proceedings of FTCS 29, Madison, Wisconsin, June 15-18, 1999.

[14] S. Nimmagadda, C. Liyanaarachchi, A. Gopinath, D. Niehaus and A.Kaushal, “Performance Patterns: Automated Scenario Based ORB Performance Evaluation”, Proceedings of COOTS 99, San Diego, California, May 3-7, 1999.

[15] Object Management Group, “The OMG’s site for CORBA and UML success stories”, http://www.corba.org/success.htm. Accessed June 22, 2000.

[16] Object Management Group, “Fault Tolerant CORBA standard”, http://www.omg.org/cgi-bin/doc?orbos/2000-01-19. October, 1999. Accessed June 19, 2000.

[17] Object Management Group, “Space Domain Task Force”, http://www.omg.org/homepages/space/index.htm. Accessed Dec 5, 2000.

[18] Object Management Group, “The Common Object Request Broker: Architecture and Specification”, August, 1997.

[19] Saib S.H., “Executable Assertions – An Aid To Reliable Software”, 11th Asilomar Conference on Circuits Systems and Computers, pp. 277-281, 1978.

[20] J. Schonwalder, S. Garg, Y. Huang, A. P. A. van Moorsel and S. Yajnik, “A Management Interface for Distributed Fault Tolerance CORBA Services”, Proceedings of the IEEE 3rd International Workshop on System Management, Newport, RI, April 1998.

[21] Kent Wilken & Timothy Kong, “Concurrent Detection of Software and Hardware Data-Access Faults”, *IEEE Transactions on Computers*, VOL. 46, No. 4, April 1997.